C/C++ Contributing Editors

The New C: Declarations and Initializations

Randy Meyers

C99 provides many new options for when and how a variable begins its life. The benefit is code that is less buggy and more readable.

Oh, April! The first stirrings of Spring and new beginnings. It is appropriate therefore that this month's column should be on the subject of initialization and declarations.

Unlike C90, C99 allows declarations to be intermixed with executable statements. Since declarations can have initializers that in some cases are run-time expressions, this raises questions about when those expressions are evaluated and when the initialization takes place.

C99 also makes two changes concerning the initialization of structs, unions, and arrays. The first change is that aggregates and unions are no longer limited to being initialized with constant expressions. The second change is a new syntax that allows you to initialize individual members or elements by name, and thus more easily initialize sparse objects and avoid the errors common to the traditional, positional brace-enclosed initializer syntax.

Constant Expressions

In this article, the term "constant expression" is going to come up several times. A constant expression is an expression whose value can be calculated before running the program. Typically, the compiler determines the value for constant expressions involving arithmetic types, and the linker or program loader determines the value of constant expressions involving addresses.

Not surprisingly, a constant expression cannot depend upon the value of any variable, since in general you must run the program to determine a variable's value. The operands of constant expressions are constants, enumeration literals, **sizeof** where the operand is not a variable length array, or addresses of functions or static variables. (Such addresses are constant by the time the program is loaded into memory.)

A constant expression cannot have side effects, and so cannot use the assignment, increment, decrement, function call, or comma operators, unless the operators are the operand of a **sizeof** operator, since **sizeof** does not evaluate the expression that is its operand.

There are places (particularly during initialization) where C requires expressions to be constant expressions. As discussed below, C99 requires that an expression be a constant expression in fewer places than C90.

Mixed Declarations and Statements

C99, like C++, does not require that all declarations appear at the start of a block. The items declared by a declaration are not visible until after their declaration, and so cannot be used by any statements or declarations that precede it in the block.

Objects with static storage duration (objects declared with the **static** or **extern** keywords or declared at file scope) must be initialized with constant expressions. Such objects are initialized once, which happens before the program starts. (C++ treats these initializations the same way, but also permits run-time initializers with different rules.)

Objects with automatic storage duration (objects declared local to a function or block without the **static** or **extern** keywords) are not initialized until their declarations are reached. In fact, the declarations act like assignment statements and evaluate the initializer expression each time the declaration is reached and assign the value to the object declared. If the same declaration declares multiple initialized objects, the "assignments" occur in the order the objects are declared. (C++ also behaves this way.)

Consider Example 1 below. The program prints the values **0**, **2**, and **4** for **j**; and **0**, **6**, and **12** for **k**.

```
// Example 1
#include <stdio.h>

int main()
{
  int i = 0;

loop:
  // j and k are initialized each
  // time their declarations are
  // reached
  int j = 2*i, k = 3*j;
  printf("%d %d\n", j, k);
  if (++i < 3) goto loop;

  return 0;
}
```

Now for some advice that many of you will resist: when possible, turn the first assignment to a variable into an initialized declaration of that variable. If you delay declaring a variable until it is needed and you can initialize it with its first value, you can eliminate uninitialized variable bugs. Many C programmers hate this advice when they first hear it. (I know I did.) The usual reasons for resisting are habit and readability. Habit ("I always did it this way") is poor justification for even an occasional bug. As to readability, this style is already common among C++ and Java programmers, and is quite readable once learned.

Of course, programs are not only art but also commercial artifacts. If you need your program to compile on platforms whose compilers do not support this C99 feature,

you have a good excuse not to use it. However, be aware that this excuse is only a temporary one until the feature is more widespread.

Non-Constant Initializers

C90 required all the expressions in a brace-enclosed initializer list for a struct, union, or array to be constant expressions. For example,

```
void f()
{
  // line below is bad C90, but good C99
  int array[2] = {g(), h()};
}
```

In C90, the initialization of **array** is invalid, since function calls are not constant expressions. C99 removes this restriction for objects with automatic storage duration and allows any run-time expression to be used in the brace-enclosed initializer for such an object. Objects with static storage duration, regardless of type, still must be initialized with constant expressions in C99.

C99 does not require that the expression in a brace-enclosed initializer list be evaluated in any particular order. In the last example, function **g** might be called before **h**, or **h** might be called before **g**. Be careful if your initializer list expressions have side effects, since you will not know the order in which they occur.

Designated Initializers

A new feature, designated initializers, permits you to name the particular member or array element being initialized. For example:

```
struct S1 {
  int i;
  float f;
  int a[2];
};

struct S1 x = {
  .f=3.1,
  .i=2,
  .a[1]=9
};
```

As you can see, designators have two different forms. The first form is **.member-name** to indicate a member of a struct or union. The second form is **[constant-expression]** to indicate an array element. Just as you can combine multiple **.** and **[]** operators in expressions, you may combine multiple designators to indicate a member of a member, a member of an array element, an array element of a member, or an array element of an array element, and so on.

In C, expressions written using the **.** and **[]** operators usually begin with an identifier that is the name of the object against which the **.** and **[]** operators are applied. Unlike expressions, designators do not begin with an indication of the object against which the **.** and **[]** designators are applied; designators are applied against the current object being initialized.

The current object being initialized changes if you nest brace-enclosed initializers within brace-enclosed initializers. The outermost brace-enclosed initializer initializes the complete object being declared, and any designated initializers that appear at that level of nesting are relative to the complete object. A nested brace-enclosed initializer whose position corresponds to an element or member whose type is array, struct, or union initializes that nested array, struct, or union subobject, and any designated initializer at that level of nesting is relative to that subobject.

Normally, each initializer in an initializer list initializes the next element or member of the object. However, designated initializers permit you to initialize the elements or members in any order. You may mix using and not using designators for the initializers. If an initializer without a designator follows an initializer with a designator, the element or member initialized by the initializer without the designator is the element or member that would normally follow the one specified by the designator.

Consider Example 2 below. All of the arrays in Example 2 are initialized to the same value. The use of designated initializers for **a3** permits the initializers to be given in reverse order. The initialization of **a4** illustrates how nested brace-enclosed initializers affect the current object to which designators are relative. The initialization for **a5** shows the mixing of initializers with and without designators.

```
//Example 2

struct S2 {
  int x, y;
};

// Arrays a1, a2, a3, a4, and a5 are
// initialized to the same values.

struct S2 a1[3] = {1, 2, 3, 4, 5, 6};

struct S2 a2[3] =
{
  {1, 2},
  {3, 4},
  5, 6
};

struct S2 a3[3] =
{
  [2].y=6, [2].x=5,
  [1].y=4, [1].x=3,
  [0].y=2, [0].x=1
};

struct S2 a4[3] =
{
  // Current object is all of a4
  [0].x=1, [0].y=2,
  {
     // Current object is a4[1]
```

```
      x=3, .y=4
  }
  // Current object is all of a4
  // Current position is [2]
  5, [2].y=6
};

struct S2 a5[3] =
{
  // After [2].x follows [2].y
  [2].x=5, 6,
  // After [0].x follows [0].y
  [0].x=1, 2,
  // after [0].y is [1]
  3, 4
};
```

You are permitted to initialize the same subobject more than once in the initializer for an object. The last initializer in the initializer list for the subobject overrides previous initializers. If you fail to initialize a subobject, the subobject is initialized with zeros of the appropriate types, as if the subobject was an object with static storage duration without an explicit initializer. In the following example, **a6[0]** and **a6[2]** are both initialized to **0**, and **a6[1]** is initialized to **100**.

```
int a6[3] = {[1]=5, [1]=100};
```

In a similar fashion, the square matrix **a7** is initialized with all zeros except for ones along the diagonal.

```
double a7[3][3] =
  {[0][0]=1., [1][1]=1., [2][2]=1.};
```

You may use designated initializers to initialize an array of unknown size. The size of the array will be determined by the highest numbered element that has an explicit initializer. In the following example, **a8** has 100 elements (maximum index value of 99); and **a8[0]** is initialized with **50**, **a8[1]** is initialized with **51**, **a8[99]** is initialized with **149**, and all other elements are initialized with zero.

```
int a8[] = {[99]=149, [0]=50, 51};
```

Designated initializers are particularly useful when initializing a union. Under C90, if you initialized a union, the initializer was for the first member of the union, and there was no way to specify that the initializer was intended for a different union member. Designated initializers permit you to name the union member you are initializing. In the following, **x.f** is initialized to **3.14**.

```
union {int i; float f;} x =
   {.f=3.14};
```

Summary

C99 removes some of the restrictions in C90. Declarations need not be grouped at the beginning of a block. Instead they may appear at the point where the item declared is actually needed. Initialization of variables with automatic storage duration acts like assignment, and more such initializations can use run-time

expressions rather than constant expressions compared to C90. This promotes program correctness by allowing the elimination of uninitialized variables.

C99 also adds designated initializers, which allow the programmer to express more explicitly which members or elements are initialized, removes the dependency on initializers appearing in a particular order, and allows initializer lists to be more succinct by only containing initializers for non-zero subobjects.

Randy Meyers *is consultant providing training and mentoring in C, C++, and Java. He is the current chair of J11, the ANSI C committee, and previously was a member of J16 (ANSI C++) and the ISO Java Study Group. He worked on compilers for Digital Equipment Corporation for 16 years and was Project Architect for DEC C and C++. He can be reached at* ***rmeyers@ix.netcom.com****.*